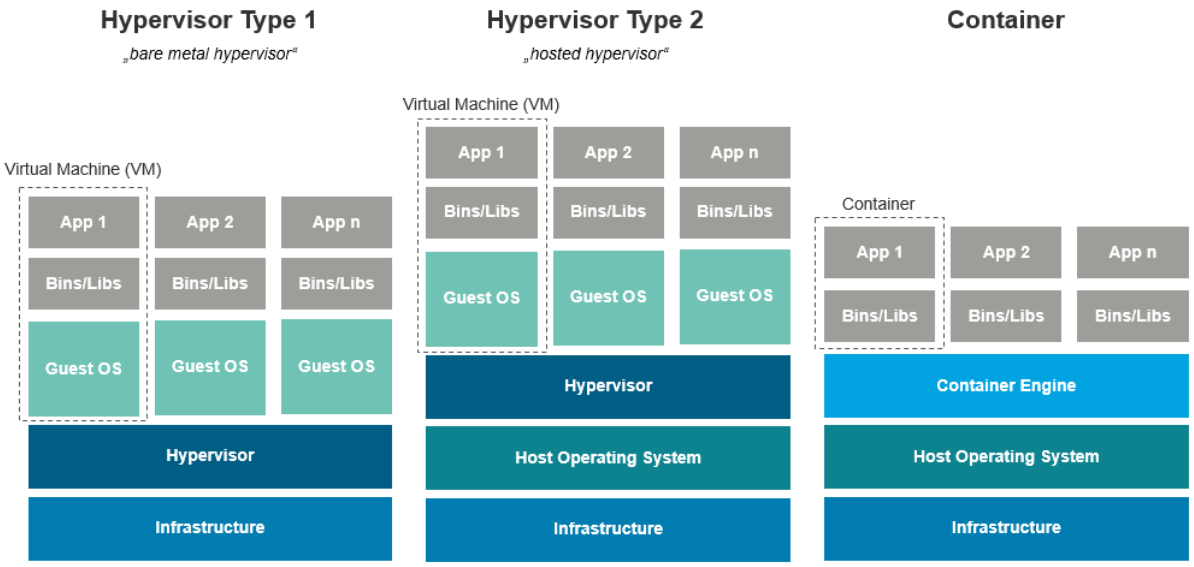
Docker

Struttura

L’architettura di docker è di tipo client-server.Tra i componenti troviamo il Docker Client e il Docker Daemon . Il Docker Client contatta il Daemon utilizzando il comando **docker** che sfrutta delle REST API , è il metodo più utilizzato per comunicare con il docker daemon ed eseguire container , un altro metodo è Docker desktop.
Il docker daemon invece, riceve ed elabora le chiamate da docker client e si occupa di costruire, eseguire e distribuire i nostri container.
La comunicazione tra il docker daemon e il docker client avviene con l’uso di REST API tramite il canale UNIX SOCKET (permette la comunicazione tra componenti dello stesso sistema) o grazie ad un’interfaccia di rete (ad esempio TCP, questo viene utilizzato per le gestione distribuita di container , potendo far comunicare il docker daemon con un client facente parte di un altro sistema).



A differenza delle VM i Container sono leggeri e condividono il kernel dell'host con altri container. Ciò significa che condividono le risorse del sistema operativo dell'host e sono più efficienti in termini di utilizzo di risorse rispetto alle VM.

Containers Docker

Immagini

Per parlare di container docker dobbiamo aver chiaro cosa sia un immagine.
Un immagine è un template di sola lettura contenente le istruzioni per creare e configurare un container docker.
Si possono trovare molte immagini già pronte all' interno dei docker registry come Docker Hub , oppure si possono creare le proprie , spesso partendo da immagini gia esistenti e aggiungendo le configurazioni di cui abbiamo bisogno.
Per creare un' immagine si utilizza un Dockerfile , un file di configurazione contenente le istruzioni per la creazione del container, ogni istruzione nel container corrisponde ad un layer durante la costruzione del container una cosa molto utile è che quando andiamo a modificare un dockerfile ed eseguiamo il rebuild dell'immagine, solo i layer che sono stati modificati vengono ricostruiti.

Containers

Un container è un'istanza eseguibile di un'immagine. Puoi creare, avviare, fermare, spostare o eliminare un container usando l'API o la CLI di Docker. Puoi connettere un container a una o più reti, allegare archiviazione ad esso o persino creare una nuova immagine basata sul suo stato attuale.

Quando avvii un container , vengono eseguite diverse azioni , se per esempio volessimo avviare un container usando il comando **docker run -it ubuntu /bin/bash**
in ordine verrebbero eseguite le seguenti azioni :

- Se non abbiamo l immagine in locale viene scaricata dal registro configurato
- Docker crea un container
- Docker assegna un filesystem di scrittura/lettura isolato al container in base alle istruzioni presenti nel dockerfile, in modo che si possano creare/modificare file e directories al suo interno mentre è in esecuzione
- Viene assegnata al container un'interfaccia di rete collegata a quella principale che gli permette di collegarsi all'esterno utilizzando la connessione internet del sistema host.
- Docker avvia il container ed esegue /bin/bash .

La separazione dei container docker è gestita dai "namespaces", una caratteristica del kernel Linux che consente di isolare e separare diversi aspetti di un sistema operativo.

I container possono essere utilizzati per lo sviluppo in modo da avere ambiente di sviluppo, trasportabile e replicabile , autoconsistente .
Queste permette agli sviluppatori di una determinata applicazione di lavorare all'interno di ambienti di sviluppo uguali , evitando problemi di dipendenze dei pacchetti e configurazioni.
I container utilizzati per questi scopi sono chiamati **dev container** e spesso utilizzano filesystem più estesi , contenenti tutto ciò di cui gli sviluppatori hanno bisogno per lavorare.
Quando si porta un applicazione in produzione , invece, si utilizzano docker images minimali contenenti solamente i binari e le librerie necessarie a far funzionare l'applicazione e

garantire il servizio. Questo diminuisce i rischi dovuti ad attacchi informatici in quanto le librerie e programmi presenti nel container sono ridotti al minimo e questo facilità la manutenzione permettendoci di esporre meno vulnerabilità possibile da sfruttare all'attaccante.

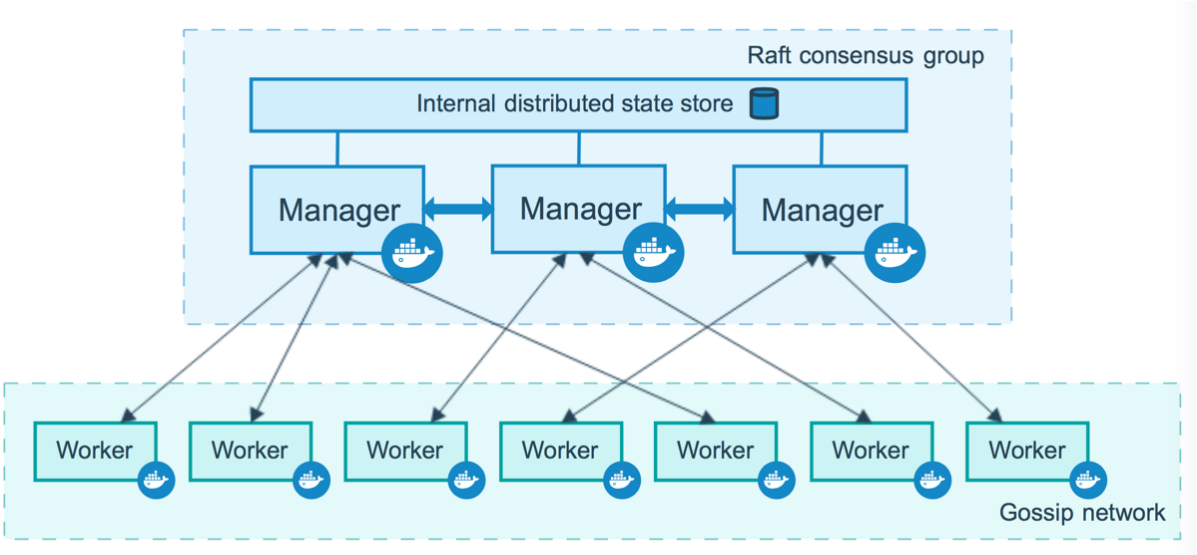
Docker Compose

Compose è uno strumento per definire ed eseguire applicazioni multi-container in Docker. Con compose si utilizza un file YAML per configurare i servizi di un'applicazione. Poi, con un singolo comando, si creano e avviano tutti i servizi dalla configurazione.
Se per esempio avessimo bisogno di sviluppare un'applicazione che necessità di un database e un ngnix server potremmo con un unico file YAML creare due container contenenti uno il servizio mysql e l'altro ngnix e farli comunicare .
Tuttavia compose è uno strumento utilizzato per lo sviluppo e il collaudo , il suo utilizzo per la produzione è sconsigliato.

Docker Swarm

Docker Swarm è una funzione docker che permette l'orchestrazione dei container attraverso la funzionalità di clustering host nativa di docker , ovvero la distribuzione dei container su più macchine fisiche che lavorano assieme.
Un cluster Swarm è formato da due tipi di nodi (i nodi sono le macchine che svolgono le attività):

- **Nodi Manager:** i nodi incaricati della gestione dei nodi di lavoro (o nodi worker) ed eseguono tutti i comandi dell'interfaccia a riga di comando di Docker relativi alla gestione e monitoraggio di uno Swarm
- **Nodi Worker** : I nodi Worker sono quelli che si occupano dell'esecuzione dei container per i servizi containerizzati



I Cluster vengono usati per distribuire i servizi containerizzati su più Host , garantendo un servizio di alta affidabilità e con un elevata scalabilità Questo è un modello molto semplice di cluster , i cluster kubernetes possono essere molto più complessi.

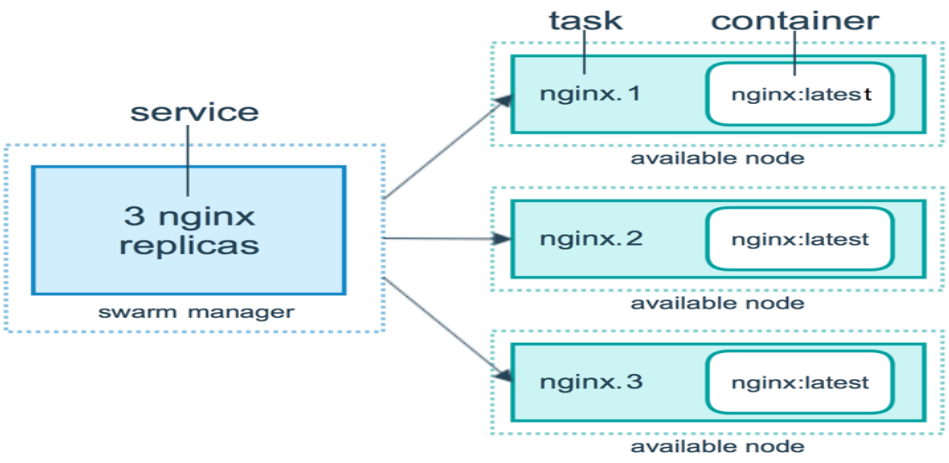
Docker Swarm si avvale di **Raft** , un protocollo di consenso distribuito , ecco alcune delle sue caratteristiche chiave:

- **Leader Election (Elezione del Leader)**: In RAFT, un nodo all'interno del cluster viene eletto come leader. Il leader è responsabile della gestione delle operazioni di scrittura e dell'invio dei dati agli altri nodi. Gli altri nodi sono chiamati "follower" e seguono le istruzioni del leader.
- **Log Replication (Replicazione del Registro)**: Ogni modifica di stato all'interno del cluster viene registrata in un registro replicato. Il leader è responsabile della replica dei dati all'interno del registro in modo che tutti i nodi abbiano una copia coerente delle modifiche.
- **Safety (Sicurezza)**: RAFT è progettato per garantire la sicurezza dei dati. Un'operazione di scrittura non viene considerata confermata finché il leader non ha ricevuto una conferma dalla maggioranza dei nodi follower.

Per distribuire un'applicazione quando Docker Engine è in modalità Swarm, si crea un **service**. Spesso, un service rappresenta l'immagine per un microservizio all'interno del contesto di un'applicazione più ampia. Esempi di servizi potrebbero includere un server xHTTP, un database o qualsiasi altro tipo di programma eseguibile che si desidera eseguire in un ambiente distribuito.

Quando si crea un servizio, si specifica quale immagine del container utilizzare e quali comandi eseguire all'interno dei container in esecuzione. Si definiscono anche le opzioni per il servizio, tra cui:

- la porta attraverso cui il servizio è reso disponibile al di fuori del cluster Swarm
- una rete sovrapposta per il servizio per connettersi ad altri servizi nel cluster Swarm
- limiti e riserve di CPU e memoria
- una politica di aggiornamento continuo
- il numero di repliche dell'immagine da eseguire nel cluster Swarm



Quando si distribuisce il servizio all'interno del cluster Swarm, il manager Swarm accetta la definizione del servizio come stato desiderato per il servizio. Successivamente, pianifica il servizio su nodi all'interno del cluster come uno o più **task** (attività replica). Le attività vengono eseguite in modo indipendente l'una dall'altra su nodi all'interno del cluster.

Storage

Lo storage dei dati in Docker è gestito attraverso un "file system a strati", noto anche come "layered file system". Questo sistema offre un approccio efficiente alla gestione dei dati all'interno dei container. Ogni immagine del container è composta da una serie di strati, ognuno dei quali rappresenta una modifica o un aggiornamento rispetto allo strato precedente. Questi strati sono di sola lettura e sono condivisi tra i container basati sulla stessa immagine, il che consente di risparmiare spazio su disco.

Quando si avvia un contenitore, Docker crea un nuovo strato di scrittura (writable layer) che si sovrappone agli strati di sola lettura dell'immagine di base. Questo strato di scrittura è specifico per il contenitore e consente al contenitore di apportare modifiche al sistema file all'interno del suo spazio di esecuzione senza influire sugli altri contenitori o sugli strati di sola lettura delle immagini.

Le modifiche apportate dal contenitore vengono memorizzate nel suo layer di scrittura, garantendo l'isolamento dei dati. Il risultato finale è un "merged layer" visibile al contenitore, che combina gli strati di sola lettura dell'immagine con il layer di scrittura specifico del contenitore. Questo merged layer costituisce il file system che il contenitore "vede" e utilizza durante la sua esecuzione. In questo modo, Docker offre una gestione dei dati efficiente e isolata per i contenitori, consentendo loro di condividere dati comuni tra immagini e contenitori, ma anche di mantenere dati specifici e modifiche isolate all'interno di ciascun contenitore.

Lo **Storage driver** utilzzato di default da docker è overlay2 , che funziona proprio come sopra citato.

I layer di un container sono contenuti all'interno della directory /var/lib/docker/overlay2
e si possono visualizzare con il comando **docker container inspect <nome container>.**

Debootstrap

È uno strumento utilizzato principalmente in sistemi basati su Debian (come Ubuntu) per creare un sistema file di base o una radice di sistema per una distribuzione Debian. Viene spesso utilizzato per la creazione di chroot environments, container Linux minimali o immagini di sistema di base per l'installazione o la personalizzazione di sistemi Debian-like.

La correlazione tra debootstrap e Docker può essere vista nel fatto che è possibile utilizzare debootstrap per creare una distribuzione Linux di base all'interno di un contenitore Docker. Ad esempio, è possibile utilizzare `debootstrap` per creare un sistema Debian di base all'interno di un container Docker, che può poi essere usato come punto di partenza per creare un'immagine Docker personalizzata. Questa immagine personalizzata può quindi essere utilizzata per eseguire i tuoi servizi o applicazioni in un ambiente containerizzato.

La compilazione Multistage

La compilazione multi stage permette di affrontare la fase di build in più fasi e la convenienza da questa scelta è semplice , ci permette di andare in produzione con un container completo di cio che serve per esporre un determinato servizio ma con la possibilità di rimuovere tutte risorse necessarie nella fase di creazione ma non a sostenere il servizio una volta creato , ad esempio :

```
docker build --tag=buildme .
docker images buildme
REPOSITORY   TAG            IMAGE ID       CREATED       SIZE
buildme          latest   c021c8a7051f   5 seconds ago   150MB
```

in questa compilazione non è stata usata la compilazione multistage e l'immagine ha un peso di 150MB , ma siccome il container viene creato utilizzando i file binari eseguibili non è necessario tenere al suo interno le risorse Go che sono state utilizzate per crearlo .

```
 # syntax=docker/dockerfile:1
 FROM golang:1.20-alpine
 WORKDIR /src
 COPY go.mod go.sum .
 RUN go mod download
 COPY . .
 RUN go build -o /bin/client ./cmd/client
 RUN go build -o /bin/server ./cmd/server
+
+ FROM scratch
+ COPY --from=0 /bin/client /bin/server /bin/
 ENTRYPOINT [ "/bin/server" ]
```

```
docker build --tag=buildme .
docker images buildme
REPOSITORY   TAG            IMAGE ID       CREATED       SIZE
buildme          latest   436032454dd8   7 seconds ago   8.45MB
```

se invece utilizzo la compilazione multi stage posso utilizzare le risorse e librerie Go per compilare i file creare gli eseguibili e poi eliminarle visto che non mi servono per mantenerli , tenendo all'interno del container solo gli eseguibili , il risultato è un container che sostiene lo stesso servizio ma con una dimensione di soli 8,46MB invece che 150MB.

Creazione Docker Swarm

Docker Swarm è un orchestratore di container docker per lo sviluppo in cluster, ci permette di creare un cluster di macchine che lavoreranno insieme per lo sviluppo di servizi scalabili
Abbiamo visto che per creare un docker swarm si esegue il comando

docker swarm init –advertise-addr <ip:porta>

Questo comando ci permette di inizializzare il nostro cluster che avrà un solo nodo manager(che sarà in automatico il nodo leader del cluster), l'opzione –asdvertise-addr è obbligatoria e indica l'indirizzo ip su cui il nodo manager si metterà in ascolto.
Eseguito questo comando possiamo utilizzare i token per aggiungere altre macchine al cluster , il comando per poter generare un token di accesso è il seguente:

docker swarm join-token manager
comando utilizzato sul nodo leader per creare un token per l'accesso di un nodo manager
docker swarm join-token worker
comando utilizzato sul nodo leader per creare un token per l'accesso di un nodo worker

una volta creato il token di nostro interesse possiamo usare il seguente comando sulla macchina che vogliamo aggiungere al cluster:

docker swarm join --token <token> <indirizzo_manager1>:<porta>
dove sostituiremo <token> con il token generato dal nodo leader.
A questo punto avremo il nostro cluster pronto per essere utilizzato .

Docker stack

Uno stack in Swarm consiste in un gruppo di servizi correlati che possono essere distribuiti e gestiti su un cluster di macchine.
Si utilizza lo stack per avere dei servizi distribuiti , scalabili e con un bilanciamento del carico.
Si crea lo stack utilizzando un file.yaml come il compose ma con delle specifiche differenti.
Le principali diciture di una versione semplificata di un file-stack sono le seguenti :

- version
- services
    - nome_service
        - image
        - environments
        - ports
        - volumes
        - networks
        - deploy
            - replicas
- networks
- volumes

```yaml
version: '3.8'
services:
  postgres:
    image: postgres:16.0-alpine3.18
    ports:
      - 5432:5433
    environment:
      - POSTGRES_USER=pippo  #sempre in /run/secrets/
      - POSTGRES_PASSWORD_FILE=/run/secrets/pg-password
    secrets:
      - pg-password
    networks:
      - network2
    volumes:
      - volume_postgres:/var/lib/postgresql/data
    deploy:
      replicas: 1

  pgadmin:
    image: dpage/pgadmin4:7.8
    labels:
      - customers=311Verona
    environment:
      - PGADMIN_DEFAULT_EMAIL=pippo@example.com
      - PGADMIN_DEFAULT_PASSWORD=password123
      - PGADMIN_LISTEN_PORT=8080
    ports:
      - 8080:8080
    volumes:
      - volume_pgadmin:/var/lib/pgadmin
    networks:
      - network2
    depends_on:
      - postgres
    deploy:
      replicas: 1

  postgrest:
    image: bitnami/postgrest:11.2.2
    environment:
      PGRST_DB_URI: postgres://pippo:ksnV93DZh9WzEA0mCDe+VDTObjg=@postgres:5432/pippo
    ports:
      - 3000:3000
    depends_on:
      - postgres
    networks:
      - network2
    volumes:
      - pgt-volume:/var/lib/postgrest/data
    deploy:
      replicas: 1



networks:
  network2:
volumes:
  volume_postgres:
  volume_pgadmin:
  pgt-volume:
secrets:
  pg-password:
    external: true
```

qui un esempio di uno stack.

L’immagine riportata definisce uno stack per lo sviluppo di tre servizi , postgres, pgadmin e postgrest .
In ogni servizio viene specificata l’immagine utilizzata per la creazione dei container che conterranno quell’applicativo.

Environments specifica delle variabili che l’applicativo utilizza , ad esempio nel servizio postgres specifichiamo
- POSTGRES_USER=
- POSTGRES_PASSWORD_FILE=
che servono a specificare un utente e la password di questo utente dentro all applicativo postgres che sviluppiamo dentro al container.

ports specifica le porte che il container esporrà e le porte della macchina host che riporteranno alla porta del container .

networks specifica la rete/le reti a cui il container sarà connesso (i networks vanno anche specificati all’interno dei service per poter essere utilizzati dai container e vanno specificati fuori dai service per essere creati).

volumes specifica il nome e la posizione dei volumi del servizio all’interno dei container (anche i volumi vanno specificati fuori dai services per poter essere creati).

deploy la voce deploy andrà a contenere delle specifiche riguardanti lo sviluppo del servizio , come ad esempio il numero di repliche che vogliamo per quel determinato servizio all’interno del cluster.

depends_on serve a specificare la dipendenza del servizio da un altro servizio , in modo da crearlo solamente quando il servizio da cui dipende è gia stato creato ed è in funzione.

Secrets, configs e labels

I **secrets** ci permettono di tenere alcune specifiche dentro a delle variabili utilizzate dai container docker senza che queste risultino in chiaro all’interno del nostro codice.
Per poter utilizzare un secret dobbiamo crearlo da riga di comando con il comando
docker secret create <nome secret> -
con questo comando ci verrà chiesto di inserire in maniera interattiva il contenuto del nostro secret.
Si puo usare anche il comando
docker secret create <nome secret> path/del/file
e utilizzare direttamente il contenuto di un file per il nostro secret .
Una volta creato il secret possiamo usarlo all’inteno del nostro file.yaml specificandolo nella sezione serets, che conterrà la lista di tutti i secret utilizzati nel nostro stack , sia all’interno del service che lo utilizzerà.Questo ci permette di utilizzare il secret al posto di un valore specifico come abbiamo fatto nel file mostrato nell’immagine di sopra , dove abbiamo usato un secret per la password del nostro utente pippo dentro a postgres.

I **configs** invece ci permettono di avere delle risorse condivise tra la macchina host e il container in esecuzione .

docker config create <nome config> <nome file>

Questo comando ci permette di creare un config dandogli un nome a nostra scelta e indicandogli la risorsa da condividere.
Come per i Secrets , una volta creati i nostri configs possiamo usarli all'interno del file.yaml specificando all'interno del service che li utilizzerà il nome del config e il path in cui vogliamo conservarlo all'interno del container, in questo modo :

```
configs:
        -source: <nome confg>
        target: /path/dentro/container
```

Le **labels** sono tag personalizzati che possono essere assegnati a container, immagini, volumi,servizi e nodi del cluster . Queste etichette forniscono un modo flessibile per aggiungere informazioni aggiuntive o identificare le risorse Docker in modo specifico per le esigenze del tuo progetto o dell'ambiente.
Un utilizzo delle labels è quello in cui , durante il deploy di uno stack , andiamo a definire che determinati servizi vengano sviluppati solamente sui nodi che detengono tale label.
Per poterlo fare dobbiamo seguire i seguenti passaggi

```
docker node update --label-add mylabel=true nodo
```

con questo comando aggiungiamo al nodo in questione una label .
Sucessivamente all'interno del docker compose file possiamo definire che un servizio venga sviluppato solo su nodi con quella label.

```
        …
        deploy:
                replicas: 2
                placement:
                        constraints:
                                - node.labels.node.worker==true
```